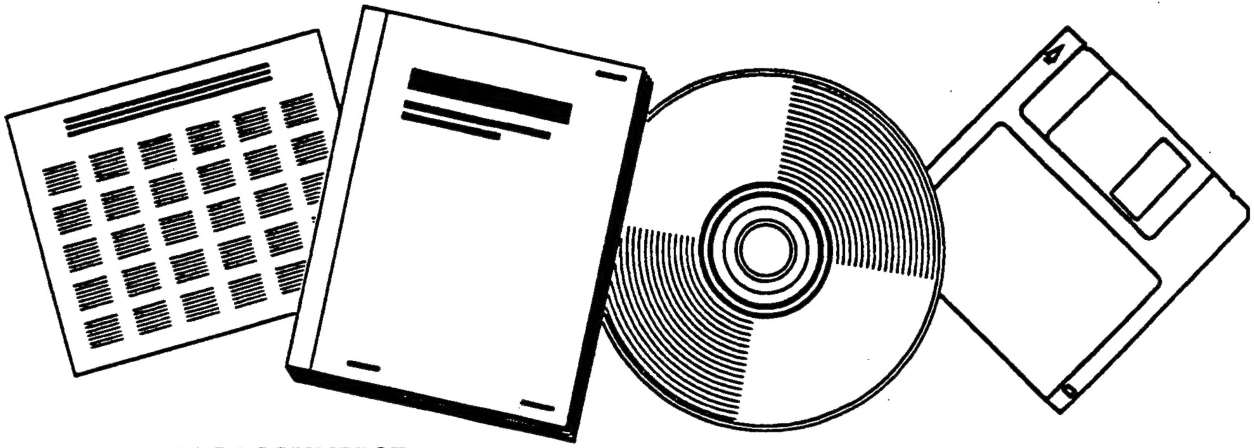
IMPLEMENTING THE PUSH-RELABEL METHOD FOR THE MAXIMUM FLOW PROBLEM ON A CONNECTION MACHINE

STANFORD UNIV., CA

FEB 92



February 1992

Report No. STAN-CS-92-1410

PB96-150404

Implementing the Push-Relabel Method for the Maximum Flow Problem on a Connection Machine

by

Farid Alizadeh and Andrew Goldberg

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PB96-150404

[illegible] :)	**2. REPORT DATE** February 1992	**3. REPORT TYPE AND DATES COVERED**
4. TITLE AND SUBTITLE Implementing the Push-Relabel Method for the Maximum Flow Problem on a Connection Machine		**5. FUNDING NUMBERS**
6. AUTHOR(S) Farid Alizadeh and Andrew Goldberg		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Computer Science Department Stanford University Stanford, CA 94305		**8. PERFORMING ORGANIZATION REPORT NUMBER** STAN-CS-92-1410
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) ONR Arlington, VA 22217		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Unlimited		**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

This paper describes an implementation of the Push-Relabel method for the Maximum Flow problem on a Connection Machine and gives computation times of the implementation on several classes of problems.

14. SUBJECT TERMS			**15. NUMBER OF PAGES** 17
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**

Implementing the Push-Relabel Method for the Maximum Flow Problem on a Connection Machine

Farid Alizadeh*
Andrew V. Goldberg†

February 1992

*Computer Science Department, University of Minnesota, Minneapolis, MN, 55455; e-mail: alizadeh@cs.umn.edu. This work was supported in part by the Air Force Office of Scientific Research grant AFOSR-87-0127, the National Science Foundation grant DCR-8420935 and the Minnesota Supercomputer Institute.

†Computer Science Department, Stanford University, Stanford, CA, 94305, e-mail:goldberg@cs.stanford.edu. This work was supported in part by ONR Young Investigator Award N00014-91-J-1855, NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT&T and DEC, Stanford Office of Technology Licensing, and a grant form Mitsubishi Corp.

1 Introduction.

In this paper we study the recent *push-relabel* [7, 11] class of algorithms for the maximum flow problem in capacitated directed networks. Several researchers [1, 3, 5, 12] concluded that their sequential implementations of push-relabel algorithms are superior to previously used codes. We focus on the behavior of the push-relabel method in massively parallel environments.

The push-relabel method is the first theoretically efficient algorithm for the maximum flow problem that is potentially practical; the previous algorithm of Shiloach and Vishkin [18], based on Dinitz' blocking flow method [6], requires the amount of memory quadratic in the number of nodes in the input network.

The push-relabel method uses two operations, *push* and *relabel*, to manipulate current flow and distance labeling functions. In the parallel implementation the algorithm pushes simultaneously from all nodes to which the push operation applies; we refer to this method of pushing as *parallel push*. Similarly, *parallel relabel* applies relabeling operation to all eligible nodes simultaneously.

The push-relabel algorithm works regardless of the order in which the push and relabel operations are applied. It also works with any correct labeling procedure. This robust nature of the algorithm allows us to experiment with various strategies for ordering push and relabel operations, and with a variety of relabeling procedures. All of the push and relabel operations are suitable for parallelization. The goal is to determine the ordering and relabel procedures most suitable for a practical parallel implementation.

This paper is a report on experiments with various strategies for the parallel push-relabel algorithm. The experiments were conducted on a Connection Machine [13] model CM-2, with 32K processors (K=1024).[1] Each processor can directly access 256K bits of local memory, so the entire system has 1 gigabytes of memory. The starting point for our work was the Connection Machine implementation of the algorithm described in [8]. Our current implementation, however, improves the original one in several areas, including better ordering and pipelining of operations of messages. Also, the original implementation was on CM-1 using the *LISP interpreter where as the current implementation is on CM-2 using the C* compiler. Our experimental results are very good on a wide class of problems; we can solve large (millions of arcs) problems in a matter of minutes.

This paper contains five sections including the introduction. In Section 2 we outline the push-relabel algorithm and discuss parallel implementation of push and various relabel operations including Derigs and Meier's gap-relabeling and parallel breadth–first search. In Section 3 we describe the mapping of networks to the Connection Machine processors. This mapping is designed to take advantage of the parallel prefix and suffix operations provided in the Connection Machine. In Section 4 we describe in more detail parallel push and relabel operations in the context of this mapping. In Section 5 we report on the running time of the program on various sample inputs and present our conclusions.

[1]We used 16K processors in our experiments because full 32K processors were often unavailable.

2 The Parallel Push-Relabel Algorithm.

In this section we review some of the basic concepts of the push-relabel method and its parallel implementation. We assume that the reader is familiar with [11]. (See also [10].) Because of high-grain parallelism of the CM-2 architecture, the goal of our implementation is to get a tight inner loop of the algorithm rather then to achieve high processor utilization by careful processor scheduling (see [9, 18]).

A *flow network* is a directed graph $G = (V, E, s, t, u)$, where V and E are node set and arc set, respectively, s and t are the source and the sink, respectively, and u is a non-negative capacity function on the arcs. We define $n = |V|$ and $m = |E|$, and assume that for each arc (v, w), the arc (w, v) is also present. A flow is a function on the arcs that satisfies capacity constraints on all arcs and conservation constraints on all nodes except the source and the sink. The conservation constraint at a node v indicates that the *excess* $e_f(v)$, defined as the difference between the incoming and the outgoing flows, is equal to zero. A *preflow* [14] satisfies the capacity constraints and the relaxed version of conservation constraints that requires the excesses to be nonnegative.

An arc is *residual* if the flow on it can be increased without violating the capacity constraints, and *saturated* otherwise. The residual capacity $u_f(v, w)$ of an arc (v, w) is the amount by which the arc flow can be increased. The residual graph is induced by the residual arcs. A residual arc (v, w) is *admissible* if $d(v) > d(w)$.

The *distance labeling* $d : V \rightarrow \mathbf{N}$ satisfies the following conditions: $d(s) = n$, $d(t) = 0$, and for every residual arc (v, w), $d(v) \leq d(w) + 1$. It is easy to see that if $d(v) < n$, then $d(v)$ is a lower bound on the actual distance from v to t in the residual graph, and if $d(v) > n$, then $d(v) - n$ is a lower bound on the actual distance of v to s.

A push-relabel algorithm maintains a preflow and a distance labeling. We say that a node v is *active* if $v \notin \{s, t\}$ and $e_f(v) > 0$. The preflow is modified using push operation, which pushes excess flow from an active node to an adjacent one that has a smaller distance label. When an active node cannot push its excess because its label is at most that of its neighbors, then the distance labeling is modified using a relabel operation, which increases the distance label of the node to the largest value allowed by the labeling constraints.

Our parallel implementation works as follows. The preflow f is initialized to $f(s, w) = u(s, w)$ for all nodes u adjacent to s, and $f(v, w) = 0$ for all $v \neq s$. The initial distance labeling is computed using breadth-first search (BFS) on the residual graph induced by the initial preflow. The preflow and distance labeling are updated using the push and relabel operations, respectively, until no active nodes remain.

The *parallel push* operation works as follows. For every active node, the parallel push operation distributes their excess among admissible outgoing arcs as described in Figure 1. In the figure,

```
OPERATION Parallel-Push

For all active nodes $v$ in parallel do
    push flow from $v$ until $e_f(v) = 0$ or $\forall w$ such that $d(w) < d(v)$, $u_f(v, w) = 0$.
```

Figure 1: The parallel push operation.

```
OPERATION Relabel

For all nodes $v \notin \{s,t\}$ in parallel do
    $d'(v) \leftarrow \min\{d(w)+1 : u_f(v,w) > 0\}$;
    If $d(v) \neq d'(v)$ then
        broadcast $d(v)$ to all neighbors of $v$;
    end;
```

Figure 2: The parallel relabel operation.

```
OPERATION gap-relabel

(1) Find the smallest $g$ where $0 < g < n$ and no node has label $g$.
(2) For all nodes $v$ in parallel do:
        if $g < d(v) < n$ set $d(v) \leftarrow n$.
(3) For all $v$ which have a new label broadcast the new labels to all neighbors of $v$.
```

Figure 3: The gap-relabel operation.

"push flow from v to w" means increasing the flow from v to w by the minimum of $e_f(v)$ and $u_f(w)$ and updating the excesses of v and w.

We use several kinds of parallel relabel operations in our implementation. The simple parallel relabel, described in Figure 2, sets a distance label of every node except s and t to one plus the minimum distance label of its residual neighbors.

Another relabeling procedure is the *gap-label* of Derigs and Meier [5] based on the following observation. Suppose at certain stage of the algorithm some nodes are labeled 0, some labeled 1, and so on, through $g-1 < n$, but no node is labeled g, although some other nodes are labeled by integers between g and n. Derigs and Meier observe that the sink is not reachable from any of the nodes whose labels d are strictly between g and n. Therefore, the labels of such nodes may just be increased to n. This procedure may easily be implemented in parallel; see Figure 3.

Note that the most accurate labeling is obtained by applying BFS backwards from the sink and forward from the source. This can be implemented by applying parallel relabeling operations until distance labels stop changing. Doing this during every pulse is too expensive. The implementation of [8] performs a breadth-first search once after a sequence of pulses so that the breadth-first search time can be amortized over the pulse time. We experimented with this approach as well as with using the gap-relabel operation instead.

No theoretical result suggests that gap-relabel and BFS operations improve the worst-case time bounds on the algorithm. In practice, both periodic BFS and gap-relabel operations reduce the number of pulses drastically. The gap-relabel operation, however, is much less expensive (usually faster than simple parallel relabel in our implementation) and as a result gives better running times most of the time despite of the fact that the labeling after such an operation is not exact.

Our parallel implementation can be viewed as running the parallel push, relabel, and gap-relabel processes "simultaneously". However, due to the SIMD nature of the machine, we have to time-multiplex the operations. We can adjust relative speed of the operations by running several push operations followed by several relabel operations followed by a gap-relabel operation. (Note that it does not make sense to run two gap-relabel operations without running a simple relabel operation in between.) We also can pipeline some of the operations as described in the next section.

A *pulse* is a sequence of pushes and relabelings (possibly pipelined) that is repeated over and over. A general rule is that the closer the labels are to their actual distance to the sink, the fewer pushes will be performed. Therefore, the more accurate labeling may reduce the total number of pulses. However, maintaining a very accurate labeling is too expensive. Our strategy is to update labels as accurately as possible without increasing the time spent in each pulse by too much.

In Section 5, we describe our implementation in more detail.

3 CM Architecture and Tools.

In this section we review some aspects of the CM architecture relevant to our program. Then, we describe a set of useful operations available on the Connection Machine for accumulating in parallel partial sums, partial minimums, etc.

3.1 Connection Machine Architecture.

The following is a brief outline of the CM architecture. For more details see the book by Hillis [13] and documents from the Thinking Machine Corporation, for instance [4].

The Connection Machine is a distributed memory parallel computer [16]. It consists of thousands of processors connected by a routing network. Each processor has local memory. The local memory of a processor can be accessed by other processors via the routing network. The CM is a *single instruction, multiple data* (SIMD) machine: the program is stored in a host computer which executes a sequential program containing parallel instructions. When a parallel instruction appears in the program, the host broadcasts it to all processors. Each processor, depending on its memory contents, either executes the instructions or remains idle. The operation of the machine is totally synchronous: the next instruction does not start until all processors have completed the execution of the current instruction.

Each processor can access its own memory, or it can access the memory of other processors. However, accessing the local memory is much faster than accessing other processors' memory. In general, the time required by a processor to access memory of a processor that is "close" (in the routing network) is less than the time required to access the memory of a processor that is far. If during the execution of an instruction several processors access memories of other processors, the longest memory access time determines the execution time of the instruction. A *routing cycle* is the amount of time it takes to execute such an instruction. It is important to realize that the routing cycle time varies depending on the interprocessor communication pattern and the size of data being accessed.

The Connection Machine software also provides the notion of *virtual processors*. The user may request any number of processors he or she wishes. If this number is larger than the number of physical processors, each physical processor simulates v virtual processors, where v is the VP ratio, that is $v = \lfloor \frac{\text{\# of virtual processors}}{\text{\# of physical processors}} \rfloor$.

3.2 Parallel Prefix and Suffix Operations.

Parallel prefix operations have been recognized as fundamental parallel operations, and their implementation and use in parallel algorithms has been widely studied; see *e.g.* [15, 17]. Given an associative binary operation "$*$" and a sequence of s numbers $a_1, a_2, \cdots, a_s$, the parallel prefix "$*$" operation maps this sequence into the sequence $a_1, a_1 * a_2, \cdots, a_1 * a_2 * \cdots * a_s$. Similarly the parallel suffix operation maps this sequence into $a_1 * a_2 * \cdots * a_s, \cdots, a_{s-1} * a_s, a_s$. In our implementation of the parallel push-relabel method, we use prefix and suffix operations with "$*$" replaced by addition, min, and copy.

An extended form of parallel prefix and suffix operations are *segmented parallel prefix* operation. In the segmented form a list of sequences $S_1, S_2, \cdots, S_l$ is mapped into $S'_1, S'_2, \cdots, S'_l$, where each sequence S'_i is obtained from the corresponding sequence S_i using the parallel prefix or suffix "$*$" operation. The lengths of sequences S_i may be different.

A sequence of numbers is stored in the Connection Machine by placing the entries of the sequence in contiguous processors. For simple operations such as addition, copying, and minimum, the time required to perform parallel prefix and suffix operations is of the same order of magnitude as the time required for one routing cycle on the data of the same size. Although the routing cycle time and the time to perform a parallel prefix operation vary depending on the communication pattern and on the type of the parallel prefix operation, we shall refer to these times as simply the routing cycle in our description of algorithms. The main point to remember is that a routing cycle is much larger than a local memory access.

4 Data Structures and Implementation Details.

4.1 Parallel Implementation of the Pulse Procedure.

We use a mapping of the input network to the machine as first described by Blelloch in [2]. This mapping allows us to use locality of data through the parallel prefix (suffix) operations. Each node v is assigned a processor P_v which we call a *node processor*. Each arc (v, w) is also assigned a processor P_{vw} called an *arc processor*. Recall that for each arc (v, w) we assume that the arc (w, v) is also present in the network and therefore, in our implementation, a separate arc processor P_{wv} is assigned to this arc. We call processors P_{vw} and P_{wv} *pair processors*. In the machine, each processor P_v is followed by all the processors P_{vw} corresponding to the arcs incident on it; the positions of arc processors associated with a node are arbitrary within themselves, and so is the positions of the node processors as long as the associated arc processors follow them. Each arc processor P_{vw} stores the processor address of its pair processor P_{wv}.

The part of the program that reads in the input, allocates processors, and initializes the system is relatively simple. The main part consists of application of the *pulse* procedure until no active nodes remain. The implementation of this procedure is summarized in Figure 4. This implementation include the simple *relabel* operations as part of it because all of the versions use this operation predominantly, although occasionally other relabel operations such as BFS or *gap-relabel* is used.

Steps 1–3 and 7 implement the parallel push procedure. In Step 1, the value of excess at each node processor P_v is sent to the arc processors immediately following it. Step 2 distributes

```
Procedure pulse

(1) For all $v \in V - \{s,t\}$ copy $e_f(v)$ to all $P_{vw}$ using seg-prefix-copy operation.
(2) { distribute excess }
    For all $P_{vw}$ use seg-suffix-add to compute the amount that can be pushed to lower
            labeled nodes through arcs that follow $(v,w)$ on the incident list of $v$.
    For all $P_{vw}$ compute the amount $\sigma(v,w)$ to be pushed from $v$ to $w$.
    For all $v \in V$ compute the amount of excess that remains at $v$ after the pushing.
(3) { Push flow }
    For all $P_{vw}$ do if $\sigma(v,w > 0$ do begin
            $f(v,w) \leftarrow f(v,w) + \sigma(v,w)$;  $u_f(v,w) \leftarrow u_f(v,w) - \sigma(v.w)$;
            send a message containing $\sigma(v,w)$ to processor $P_{wv}$.
    end.
    For all  $P_{wv}$ that received $\sigma(v,w)$ do begin
            $f(v,w) \leftarrow f(v,w) - \sigma(v,w)$;  $u_f(v,w) \leftarrow u_f(v,w) + \sigma(v.w)$;
    If simple relabel is the relabeling operation chosen then begin
(4)         { Compute new distance labels }
            For all $P_{vw}$ do
                    if $u_f(v,w) > 0$ then head-label$(v,w) \leftarrow d(v) + 1$
                    else head-label$(v,w) \leftarrow 2n$.
            For all $v \in V - \{s,t\}$ compute new-d$(v)$ using seg-suffix-min.
(5)         For all $v \in V - \{s,t\}$ copy new-d$(v)$ to all $P_{vw}$ using seg-prefix-copy
(6)         { Broadcast new labels }
            For all $P_{vw}$ such that $v \notin \{s,t\}$ do
                    If $d(v) \neq new - d(v)$ then
                       send a message containing the value of $d(v)$ to $P_{wv}$
                       and set $d(v)$ to new-d$(v)$ that was broadcast.
    end.
(7) { Update excess }
    For all $w \in V$ do begin
            Use seg-suffix-add to compute the amount of flow new-$e_f(w)$ pushed into $w$,
            $e_f(w) \leftarrow e_f(w)$+new-$e_f(w)$
    end.
```

Figure 4: Implementation of the pulse procedure.

```
OPERATION Parallel BFS

$d'(s) \leftarrow n$, and $d'(t) \leftarrow 0$;
For all nodes $v \in V - \{s,t\}$ $d'(v) \leftarrow 2n$;
Repeat
        Run steps (3), (4) and (5) of Pulse procedure.
Until $new\text{-}d(v) = d(v)$ for all nodes $v$.
```

Figure 5: Implementation of parallel breadth-first search operation.

the node excess to the outgoing arcs. First, each arc processor P_{vw} determines how much excess may be sent through its arc. This is equal to the residual capacity if $d(w) < d(v)$ (that is if w is estimated to be closer to sink) and zero otherwise. Then a *seg-suffix-add* is performed on these values. Now, each arc processor P_{vw} has information about the excess to be pushed from v, the amount it can push, and the amount that can be pushed through the arcs that follow (v, w) on the arc list of v. This information is enough to compute the amount $\sigma(v, w)$ to be pushed through the arc (v, w). After an execution of the *seg-suffix-add* operation, each node processor P_v contains the information about how much excess can be pushed from the node v at this pulse. The processor sets the value of its variable $e_f(v)$ to the amount that will remain after the pushing. Finally in step 3, all arc processors P_{vw} for which the amount $\sigma(v, w) > 0$, increase$f(v, w)$ by $\sigma(v, w)$, decrease $u_f(v, w)$ by the same amount, and send the value of $\sigma(v, w)$ to their pair processor P_{wv}. Each processor P_{wv} that receives such a message decreases $f(w, v)$ by $\sigma(v, w)$ and increases $u_f(w, v)$ by the same amount. Step 7 computes the new excesses on each node by performing a *seg-suffix-add* on the amount of new flow pushed through each arc, and this amount is added to $e_f(v)$.

Steps 4–6 implement the simple *relabel* operation. In Step 4, each processor P_{vw} sets its variable *head-label* to either $d(w)+1$ or $2n$, depending on whether the arc (v, w) is residual or not. This process involves only local memory access. Next, a *seg-suffix-min* operation is performed on the *head-label* variable, and as a result each node processor P_v contains new value of $d(v)$. In Step 5 all node processors except P_s and P_t copy this value to their corresponding arc processors using *seg-prefix-copy*. In Step 6 each arc processor P_{vw} checks if the new $d(v)$ is different from the old one and if so, sends a message to its pair processor P_{wv} updating $d(v)$ in the pair processor.

Each step 1 through 7 contains either a segmented parallel prefix (suffix), or a communication primitive, and the running time of each step is dominated by the primitive. Therefore, the overall running time of the pulse procedure is roughly seven routing cycles of the machine. Also observe that general communications are done along paths that are fixed through entire program: each processor P_{vw} has to communicate to processor P_{wv} in steps 3 and 6 and these are the only general communication operations. Therefore, the communication path has to be computed only once for each arc processor and the same information is used throughout the program.

To implement the parallel BFS operation we simply take steps 4–6 of pulse and run them over and over until the labels do not change. The number of times the simple *relabel* is iterated in a BFS operation is at most the larger of maximum distance of a node to the sink (if sink is reachable) and maximum distance of a node to the source (if sink is not reachable) in the residual graph.

The gap-relabel procedure is also easy to implement on the Connection machine; see Figure 6. Clearly this procedure is not much more costly than a simple *relabel* operation (roughly four routing cycles vs. three). However, it may increase the labels of many nodes by a substantial

```
OPERATION Parallel gap-relabel
(1) For each node whose label $d(v)$ satisfies $d(v) < n$ in parallel do:
        Broadcast a flag to the processor numbered $d(v)$;
(2) Find the smallest $g$ where processor $g$ did not receive any message in step 1).
(3) For all nodes $v$ in parallel do:
        if $g < d(v) < n$ set $d(v) \leftarrow n$.
(4) For all $v$ with new labels copy new-$d(v)$ to all $P_{vw}$ using seg-prefix-copy.
(5) For all $p_{vw}$ which received new labels send a message to $P_{wv}$ containing new-$d(v)$.
```

Figure 6: Implementation of the parallel gap-relabel operation.

amount.

We have experimented with several variants of using gap-relabel and BFS operations. In all of the variants each pulse uses pushes and simple relabels, but certain times instead of the simple relabel a BFS or gap-relabel operation is used.

When using BFS we follow the following rule. We save the amount of computational work done in the last call to BFS. Then we accumulate the amount of work done by the simple relabel since the last call to BFS. We also fix a parameter k. If k times the amount of work since the last call to BFS exceeds the amount of work in the last BFS then we use BFS, otherwise we use simple relabeling. The amount of work itself can be measured in several ways. One way is to simply look at the CPU time used. Another way is to count the number of "expensive" operations, in this case the number of routing cycles. Each simple relabel contributes three routing cycles (one parallel suffix copy, one parallel prefix min, and one general communication step), and the work accumulated by the simple relabeling procedure is simply three times the number of pulses since last call to BFS. The amount of work in each BFS varies as the residual graph changes with each new preflow. We have used this technique for both push–relabel and push–push–relabel methods. The latter is a variation of a the push-relabel method where we only relabel every other pulse.[2] (One may think of this method as choosing the relabeling operation that does nothing, and alternate using this operation with simple relabeling.) We also tested this technique with the pipelined variants of push–relabel and push–push-relabel techniques (to be discussed in the next section.)

Another approach is to use push and relabel operations but after each relabel to apply a gap-relabel operation as well. Our experiments show that one does not need to apply gap-relabel every time. We fix a parameter k and call gap-relabel after every k pulses. Again, we tested gap-relabel with both push–relabel and push–push–relabel variants of pulse and their pipelined versions. All of the timings are reported in Section 5.

4.2 Pipelining Independent Operations.

On the Connection Machine, if two segmented parallel prefix or suffix operations work on exactly the same sequences and perform the same binary operations, it is possible to *pipeline* them so that the pipelined operation, performed on the two sequences at once, is faster than two operations, each performed on one of the sequences at a time. For instance, steps 1 and 5 of the pulse could

[2] In general, a pulse can have x push operations followed by y relabel operations.

be pipelined, and so could steps 2 and 7. Steps 3 and 6 also do the same kind of operation, except that they involve message routing instructions. In theory we should be able to get better performance by pipelining the message routing steps, but on the Connection Machine we have not seen significant improvement. Pipelining the segmented prefix operations, however, results in about 10 to 20 percent improvement.

There are seven routing cycles in the pulse procedure (including cycles in simple relabeling). Some steps need to be performed after others (for instance, Step 6 must follow Step 5, and Step 5 must follow Step 4), whereas others are independent of each other (for example, steps 1 and 2 do not depend on each other and either may follow the other.) Figure 7 shows the dependency relationship among these steps.

The problem with pipelining steps 1 and 5, 2 and 7, and 3 and 6 is that they are sequentially related in the dependency graph. In order to curb this problem we take advantage of the robustness of the method. The dependency of Step 4 on Step 3 exists so that in the relabel step we have the updated residual capacities. Also, the dependency of Step 2 on Step 6 (of the previous pulse) exists so that the push operation is done based on new labels. These dependencies may be loosened somewhat so that steps 1 and 5, steps 2 and 7, and steps 3 and 6 can be pipelined. The disadvantage is that now the labeling becomes less accurate, and this translates into more iterations of the pulse procedure. The advantage is that now each pulse has only four routing cycles, three of which are pipelined (and thus operate on longer data). See Figure 8.

The question is whether the time saved in each pulse more than compensates the time lost due to increased number of pulses. We report on the experiments in the next section.

We also used pipelining on the push-push-relabel implementation. (Recall that in this implementation we call the relabel operation only every other pulse.) The dependency graph for this version of the algorithm is unfolded in Figure 9. Notice that every stage in Figure 9 is equivalent to two pulses, only one of which has a relabel stage. Thus in this form the total number of routing cycles for two pulses is seven, whereas in Figure 8 there are eight routing cycles per two pulses.

5 Experimental Results.

In this section we report on the running times of our program on several classes of medium and large networks. These experiments were conducted on two similar Connection Machines, one located at the Thinking Machine Corporation in Cambridge, Massachusetts, and the other one at the Army High-Performance Computing Research Center (AHPCRC) at the University of Minnesota in Minneapolis. Both machines have 32K processors with 1 Gigabytes of memory. The timings reported here are based on test runs on the machine in AHPCRC. The program was coded using the new C* programming language, an extension of standard C for data parallel programming.

Finding a "fair" set of input networks to test the parallel push-relabel method is not an easy task. One can easily find instances that cause the program run very slowly. For instance, since the number of pulses is at least as large as the length of the shortest path from s to t, the program will perform poorly on any graph with large $s-t$ distance. A simple path of size 64K will require 64K pulses, each taking several routing cycle of the Connection Machine, which is large compared to memory access time of a typical sequential computer. In fact, in this degenerate case only one node is active at a time, and our implementation exhibits no parallelism. On the other hand, if

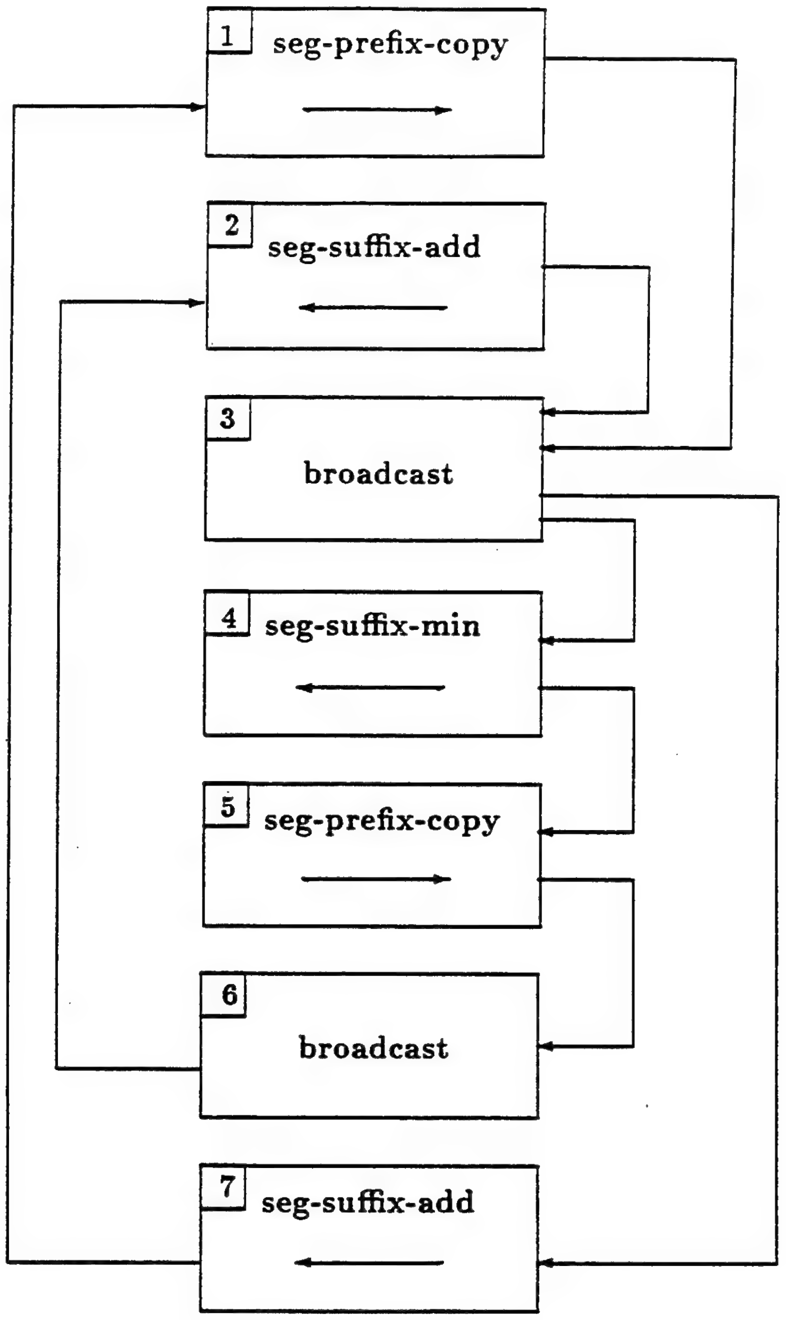


Figure 7: The dependency graph in the pulse procedure.

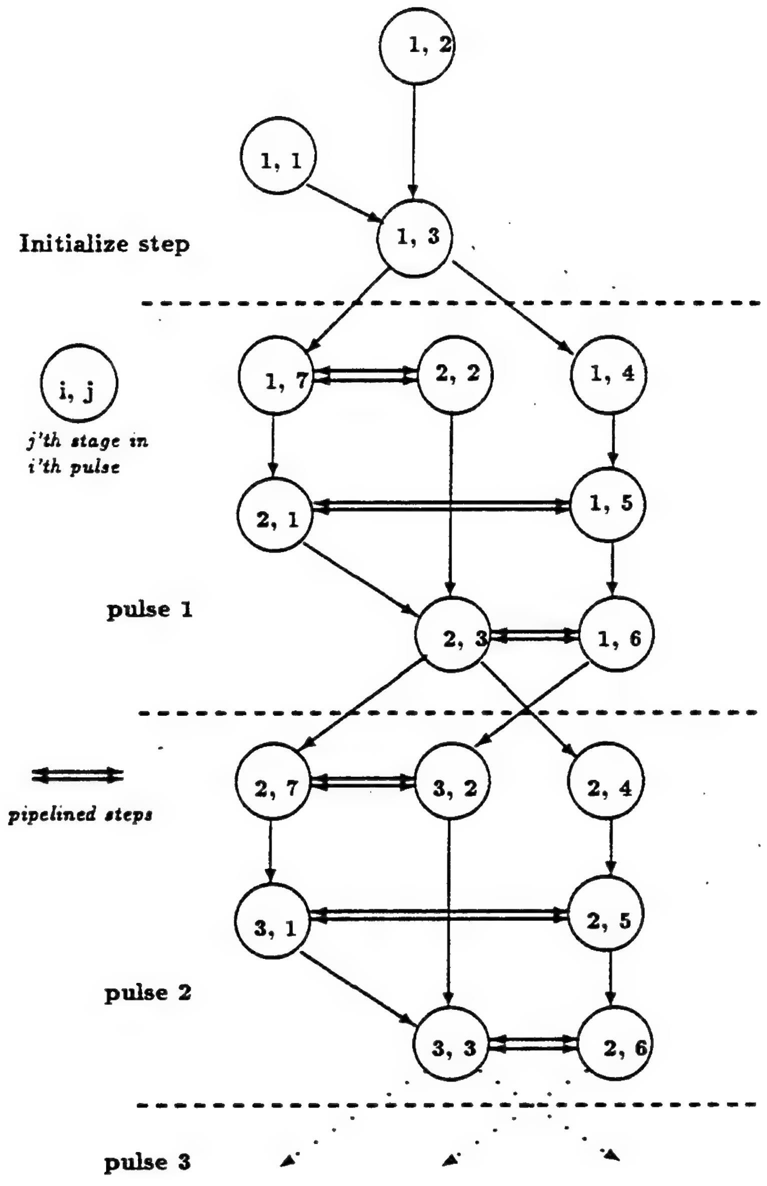


Figure 8: Unfolded dependency graph of the pulse procedure with pipelined steps.

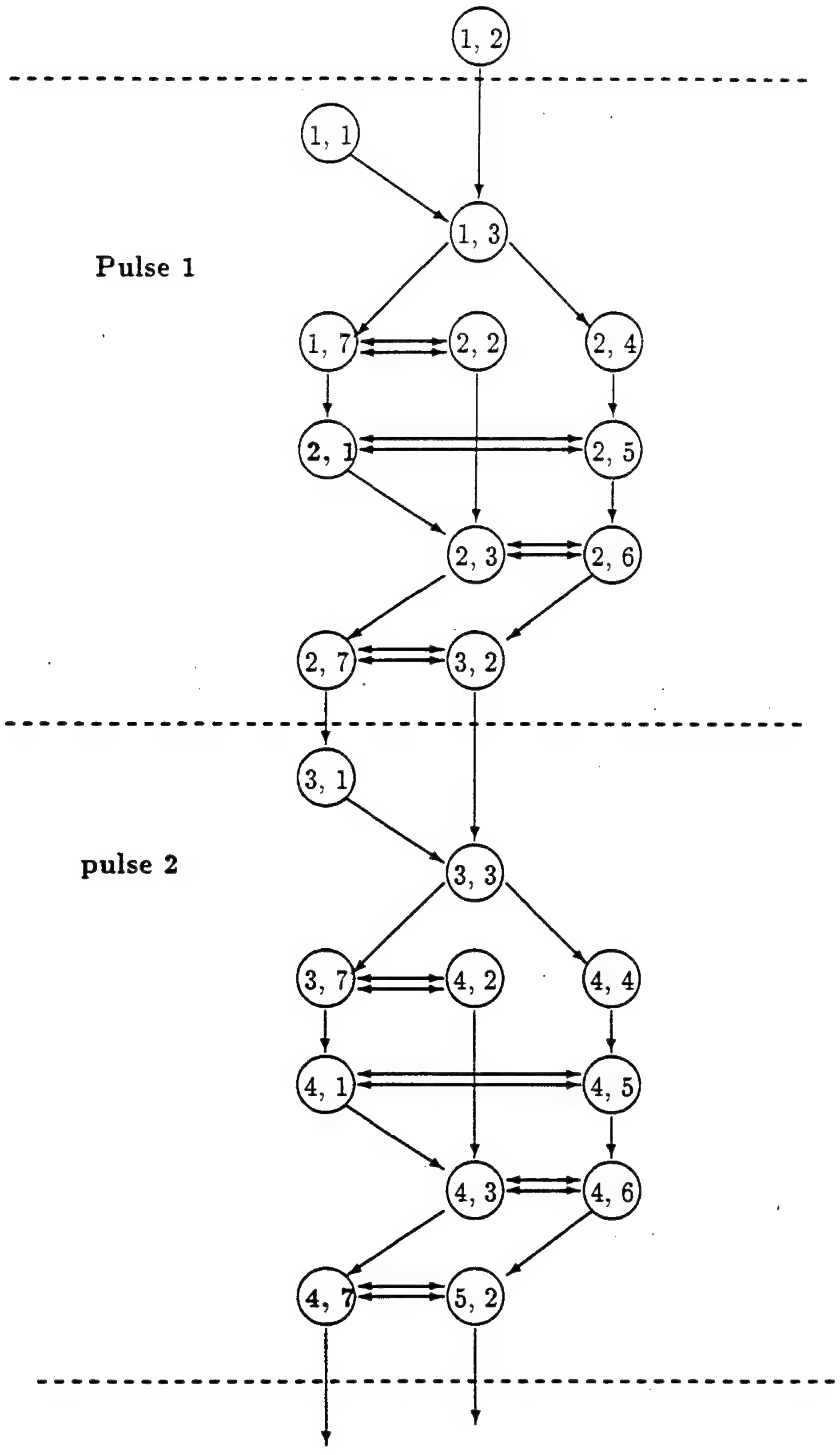


Figure 9: Unfolded dependency graph of push–push–relabel procedure with pipelined steps.

a graph consists of 64K parallel arcs connecting the source to the sink, our implementation will terminate in just one pulse.

One class of networks we used is generated as follows [8]: imagine an infinite pipe with a mesh drawn on it. Suppose the pipe goes from west to east. The distance from a node of the mesh to the nearest neighbor in both horizontal and vertical directions is one, and the circumference of the pipe is D. First we construct a graph on the nodes of the mesh. In this graph, every node has out-degree $2\delta_X + 2\delta_Y$. To construct the graph, we connect each node v by a directed arc to all nodes within δ_X due east and west, and within δ_Y due north and south. The capacity of an arc (v, w) depends on the distance x between v and x in the mesh. The capacity is selected from a uniform distribution on the interval $[0, \min(1.8^x, 10000)]$. To complete the construction, we introduce a source s and a sink t. Then we cut the pipe by two planes perpendicular to the axis of the pipe. The cutting planes are distance L apart. We consider the portion of the pipe that is between the two planes, and identify all nodes to the west of the west plane into source s and all nodes on the east of the east plane into sink t. All the arcs cut by the left plane are connected to the source, and those cut by the right plane are connected to the sink. In Figure 10 we list the node and arc sizes of *mesh-on-pipe* graphs we experimented with.

Graph	X	Y	δ_X	δ_y	nodes	arcs	size
mesh1	32	32	5	5	1026	10080	21186
mesh2	64	32	8	5	2050	25984	54018
mesh3	128	32	12	5	4098	67904	139906
mesh4	256	32	16	5	8194	168704	345602
mesh5	64	64	8	8	4098	64256	132610
mesh6	512	16	23	4	8194	217504	443202
mesh7	256	256	16	16	65538	2070528	4206594

Figure 10: Characteristics of *mesh-on-pipe* graphs.

The size parameter is larger than sum of nodes and arcs because for each arc (v, w) we had to create the pair arc (w, v). The task of reading the input is done by the front end machine (which is a SUN 4 system), and involves reading one line of input at a time, allocating a processor for the arc that was read and its pair, and sending the information (head and tail of the arc and its capacity) to the allocated processors.

Graph	a	b	nodes	arcs	size
rmf1	16	128	32768	155392	220672
rmf2	16	4	1024	4608	6400
rmf3	16	64	16384	77568	11008

Figure 11: Graphs generated by the "genrmf" generator.

We also experimented with a number of other networks provided for the DIMACS Challenge, namely the ones generated by the "genrmf" generator, fully dense acyclic networks, and some moderately sparse grids. In figures 11, 12, and 13 we list sizes of the networks that we used in our experiments.

For each of these networks we have run four basic algorithms: push-relabel without pipelining, push-relabel with pipelining, push-push-relabel without pipelining and push-push-relabel with

Graph	nodes	arcs	size
acyclic1	512	130816	262144
acyclic3	2048	2096128	4194304

Figure 12: Complete acyclic networks.

Graph	nodes	arcs	size
BLM1	16386	522211	1060808
BLM2	4098	65029	134156

Figure 13: Moderately sparse Basic Line Mesh networks.

pipelining. In all cases the simple relabel is used in the pulse procedure, except at certain pulses gap-relabel or BFS operation was is used instead, as discussed in the previous section. Gap-relabel was used every other k pulses for some parameter k. Although the best parameter depends on the structure of the input graph, we fixed $k = 10$, so that the results are comparable. For BFS, as was mentioned earlier, we compare the amount of "work" (in our case the number of routing cycles) done by the last BFS, and if it exceeds k times the accumulated "work" done by simple relabels then we use BFS. Again the optimal value of k varies depending on the structure of the graph; in our experiments we used $k = 2$ throughout. We have observed that the versions using gap-relabel procedure almost universally outperform those which use BFS. Figure 14 shows the computational results when gap-relabel procedure is used. Note that for large enough problems, our algorithm would run almost twice as fast on a 32K processors due to a lower VP ratio. (Recall that our data is for 16K processors.)

16.

In order to demonstrate the effect of changing the parameter k for both algorithms that use gap-relabel, and those that use BFS, we report the behavior of two algorithms on the network rmfl. The results for push-push-relabel without pipelining, with BFS and k changing from 2.0 to 3.0 in increments of 0.2 are shown in figure 15, while those of push-relabel, without pipelining, with gap-relabel and for k changing from 10 to 14 in increments of 1, on the same network are shown in Figure 16. Generally, we have observed that for $7 \leq k \leq 15$ for gap-relabel and $1.5 \leq k \leq 3.0$ for BFS all variants perform well. However, within these ranges there is no unique best value of k and one may see oscillations in the total number of pulses and in the CPU running time as k changes. This phenomenon may be attributed to the fact that it is not just the frequency of BFS or gap-relabel operations that affects the total number of pulses, but the instance that they are used is also important. The effect of calling such an operation is influenced by the current structure of the residual graph. It is not clear how to determine the most appropriate moment to apply the accurate relabeling without doing a lot of time-consuming operations.

6 Conclusions

We described several parallel implementations of the push-relabel method for the maximum flow problem and evaluated several heuristics that improve the practical performance of the method. Our results are impressive on some classes of problems. They also provide feedback for the designers of parallel computers.

Graph	Size	push-relabel without pipelining		push-relabel with pipelining		push-push-relabel without pipelining		push-push-relabel with pipelining	
		pulses	time	pulses	time	pulses	time	pulses	time
mesh1	21186	48	0.42	61	0.55	64	0.45	44	0.57
mesh2	54018	45	0.61	52	0.77	46	0.51	30	0.63
mesh3	139906	79	3.11	93	4.36	108	3.46	64	4.02
mesh4	345602	98	7.39	111	10.05	128	8.47	82	10.09
mesh5	132610	69	2.70	86	4.06	95	3.02	51	3.29
mesh6	443202	110	8.61	122	11.58	146	9.25	84	10.75
mesh7	4206594	124	135.67	135	176.43	152	137.54	99	176.97
rmf1	220672	1410	114.30	1497	135.94	1855	117.53	986	126.96
rmf2	6400	835	4.97	939	5.98	1084	5.25	635	5.91
rmf3	110080	1024	40.60	1164	52.55	1322	41.94	789	49.39
acyclic1	262144	368	14.11	478	22.23	471	14.75	257	16.01
acyclic3	4194304	1062	617.44	1477	1048.98	1391	658.61	808	759.86
BLM1	1060808	464	157.51	962	484.05	599	164.21	304	167.18
BLM2	134156	97	4.94	106	6.01	103	4.54	56	4.69

Figure 14: number of iterations of pulse operation and running time (in seconds) of pipelined and unpipelined push-relabel procedure with gap-relabel used every 10 pulses. For all runs 16K processors were used.

k	pulses	time
2.0	1664	136.40
2.2	1610	123.40
2.4	1620	126.01
2.6	1684	133.10
2.8	1752	136.50
3.0	1669	119.26

Figure 15: Effect of push-push-relabel without pipelining and using BFS on network rmfl.

k	pulses	time
10	1410	114.30
11	1341	107.74
12	1431	113.8
13	1374	110.69
14	1402	109.89

Figure 16: Effect of push-relabel without pipelining and using gap-relabel on network rmfl.

Acknowledgment

We would like to thank the Thinking Machine Corporation and the Army High Performance Computing Research Center at the University of Minnesota for making their Connection Machines available to us. The first author would also like to thank J. B. Rosen for his support during the course of this work, and Gene Golub and the Stanford University Computer Science department for their hospitality.

References

[1] R. K. Ahuja and J. B. Orlin. Personal communication. 1987.

[2] G. Blelloch. Parallel Prefix vs. Concurrent Memory Access. Technical report, Thinking Machines, Inc., 1986.

[3] B. V. Cherkassky. Personal communication. 1991.

[4] Thinking Machine Corporation. *Connection Machine Model CM2 Technical Summary.* 1990.

[5] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.

[6] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[7] A. V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.

[8] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

[9] A. V. Goldberg. Processor-Efficient Implementation of a Maximum Flow Algorithm. *Information Processing Let.*, pages 179–185, 1991.

[10] A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Flows, Paths, and VLSI Layout*, pages 101–164. Springer Verlag, 1990.

[11] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988. A preliminary version appeared in *Proc. 18th ACM Symp. on Theory of Comp.*, 136-146, 1986.

[12] M. D. Grigoriadis. Personal communication. 1988.

[13] W. D. Hillis. *The Connection Machine.* MIT Press, 1985.

[14] A. V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Math. Dok.*, 15:434–437, 1974.

[15] R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *J. Assoc. Comput. Mach.*, 27:831–838, 1980.

[16] C. E. Leiserson and B. M. Maggs. Communication-Efficient Parallel Graph Algorithms. In *Proc. of International Conference on Parallel Processing*, pages 861–868, 1986.

[17] J. T. Schwartz. Ultracomputers. *ACM Trans. Prog. Lang. and Syst.*, 2:484–521, 1980.

[18] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ Parallel Max-Flow Algorithm. *J. Algorithms*, 3:128–146, 1982.